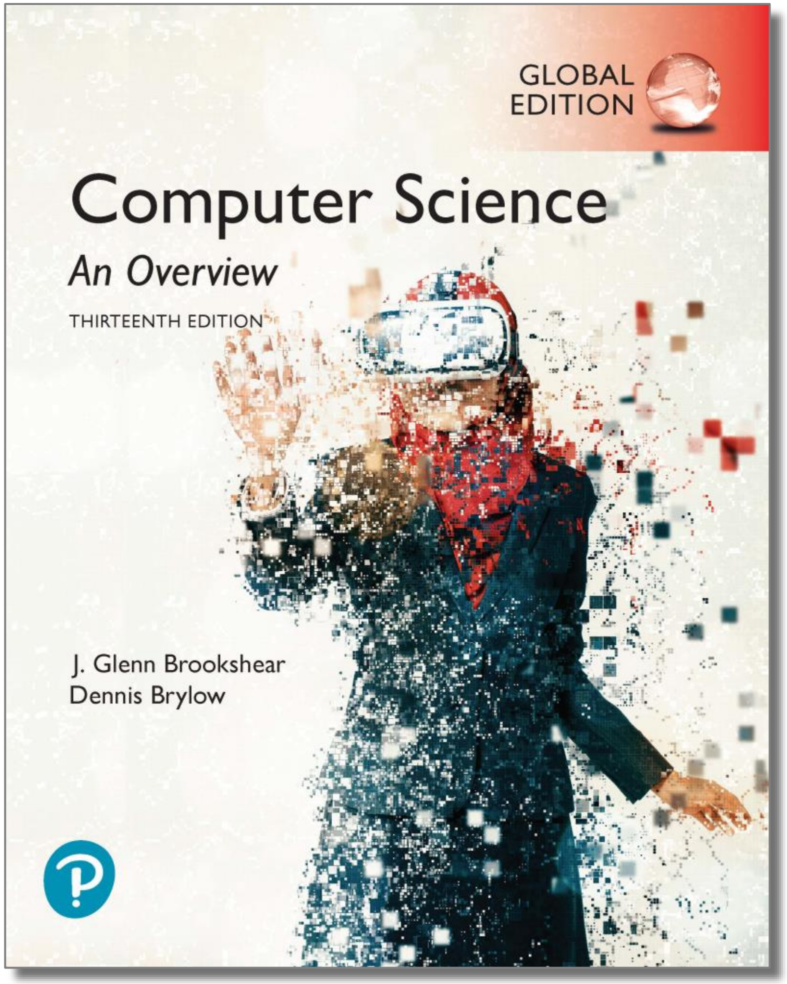
Computer Science An Overview

13th Edition, Global Edition



Chapter 8

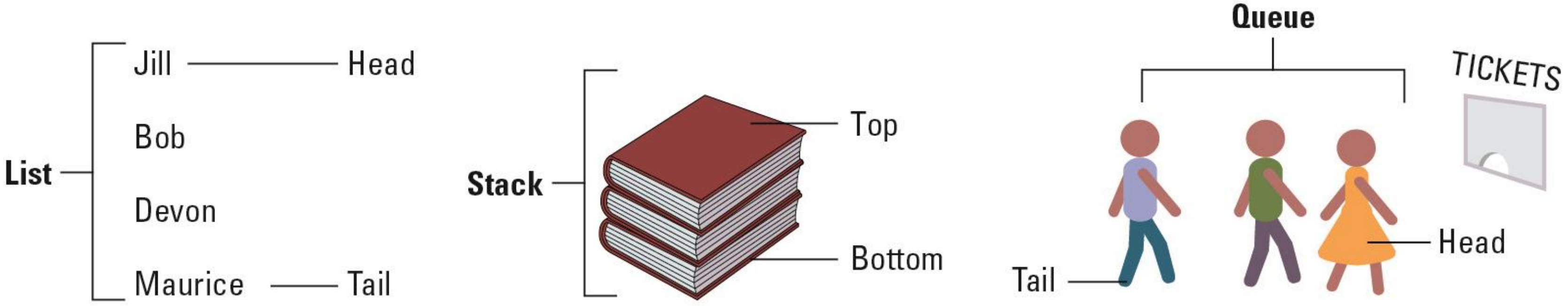
Data Abstractions

Chapter 8: Data Abstractions

8.1 Basic Data Structures

- Arrays
- Aggregates
- Lists
  - Stacks
  - Queues
- Trees

Figure 8.1 Lists, stacks, and queues



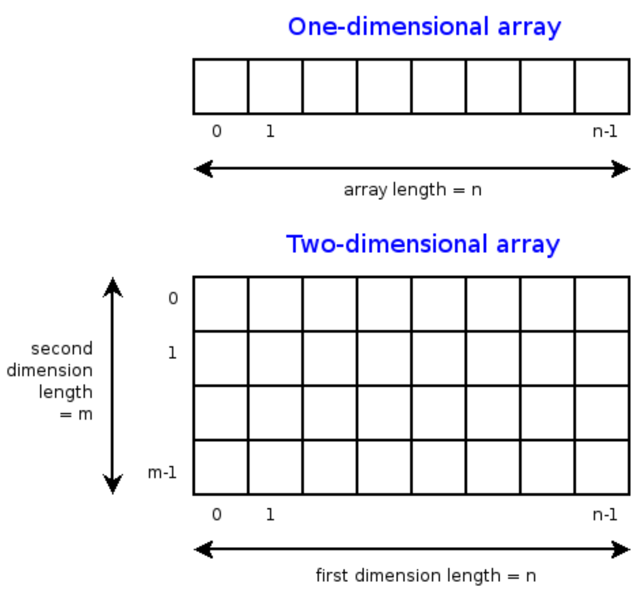
a. A list of names

b. A stack of books

c. A queue of people

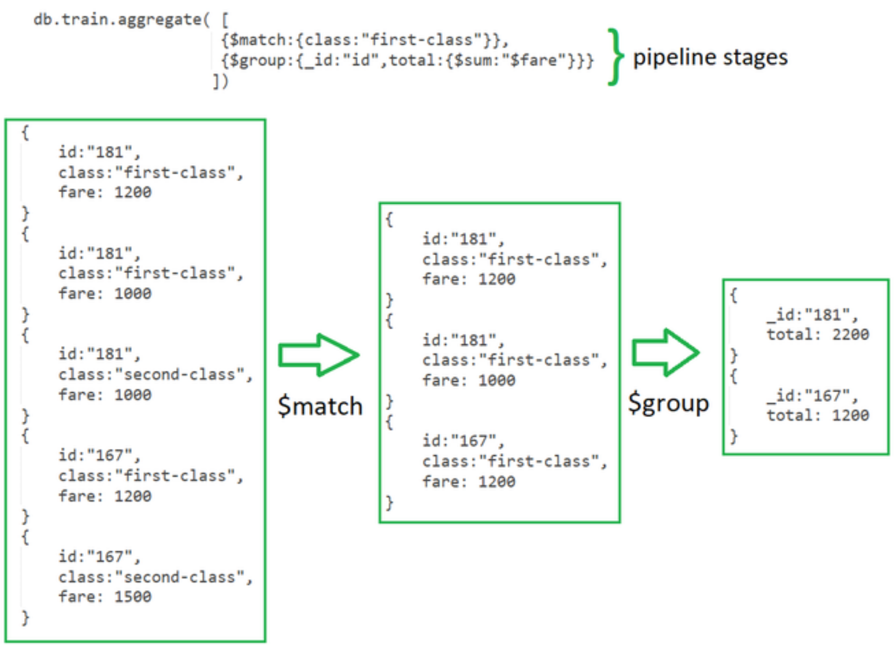
Terminology for Arrays

- **Array:** A block of data whose entries are of same type
- A two **dimensional** array consists for rows and columns
- **Indices** are used to identify positions



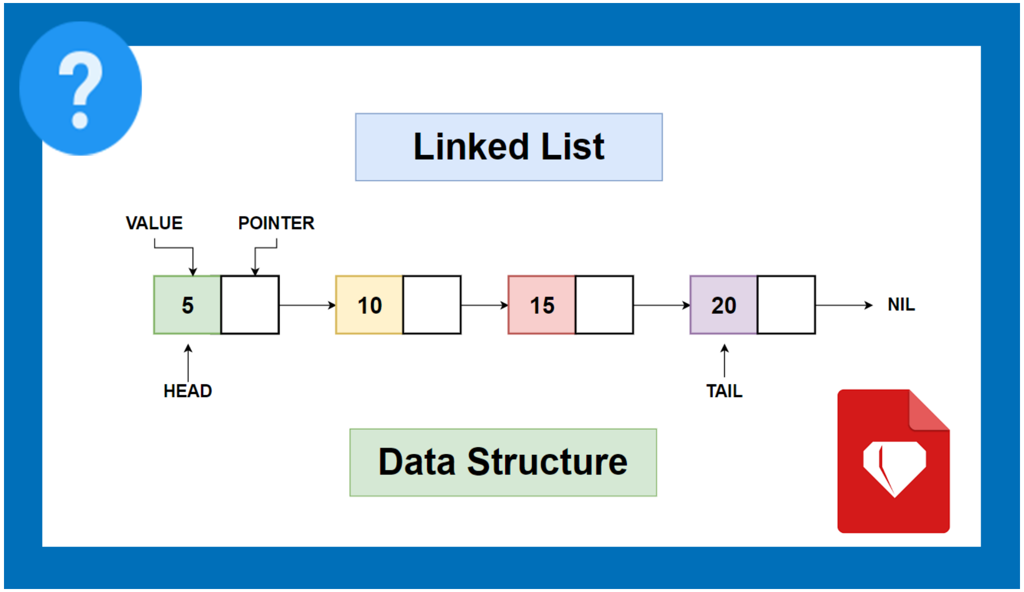
Terminology for Aggregates

- **Aggregate:** A block of data items that might be of different type or sizes
- Each data item is called a **field**
- Fields are usually accessed by name



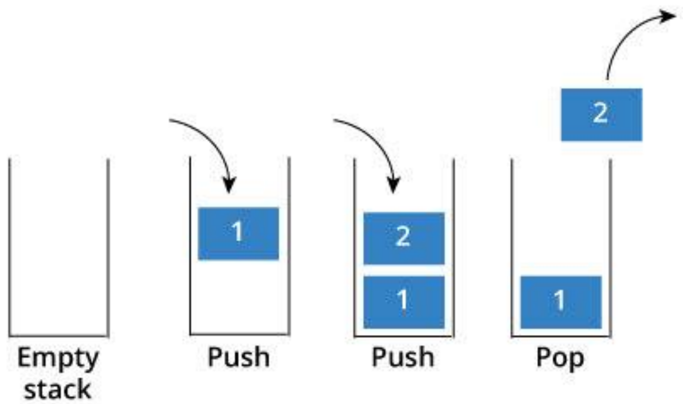
Terminology for Lists

- **List:** A collection of data whose entries are arranged sequentially
- **Head:** The beginning of the list
- **Tail:** The end of the list



Terminology for Stacks

- **Stack:** A list in which entries are removed and inserted only at the head
- **LIFO:** Last-in-first-out
- **Top:** The head of list (stack)
- **Bottom** or **base:** The tail of list (stack)
- **Pop:** To remove the entry at the top
- **Push:** To insert an entry at the top



Terminology for Queues

- **Queue:** A list in which entries are removed at the head and are inserted at the tail
- **FIFO:** First-in-first-out

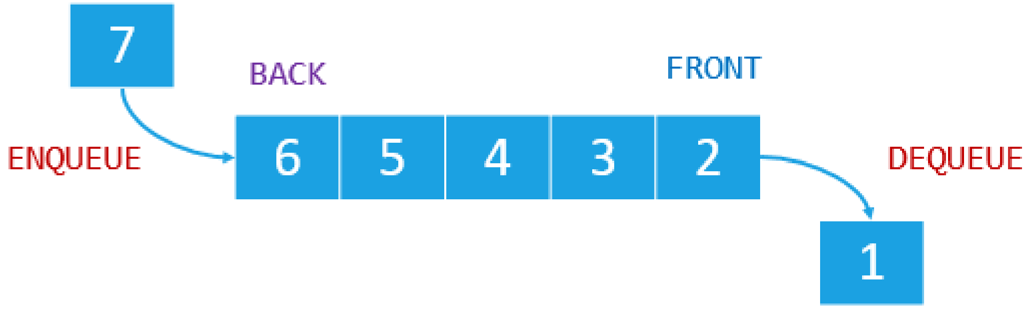
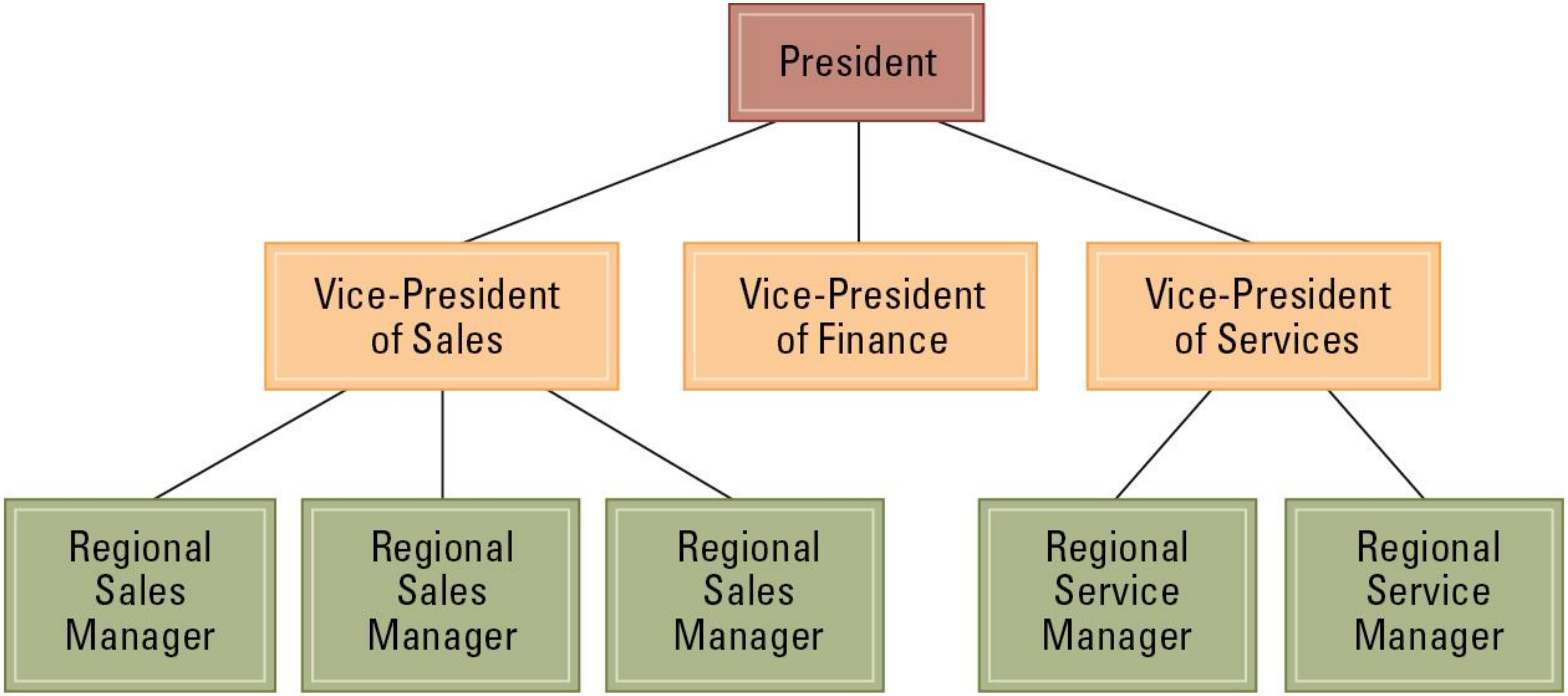
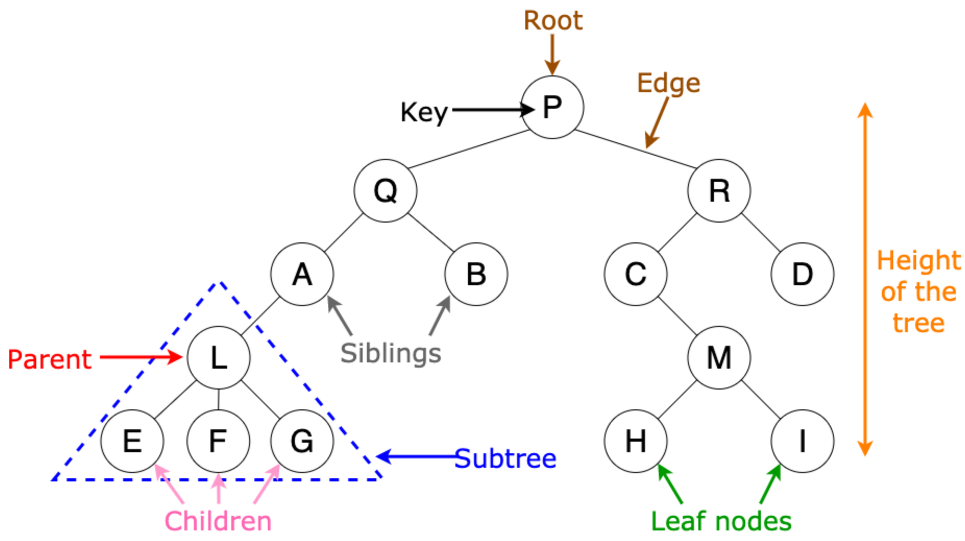


Figure 8.2 An example of an organization chart



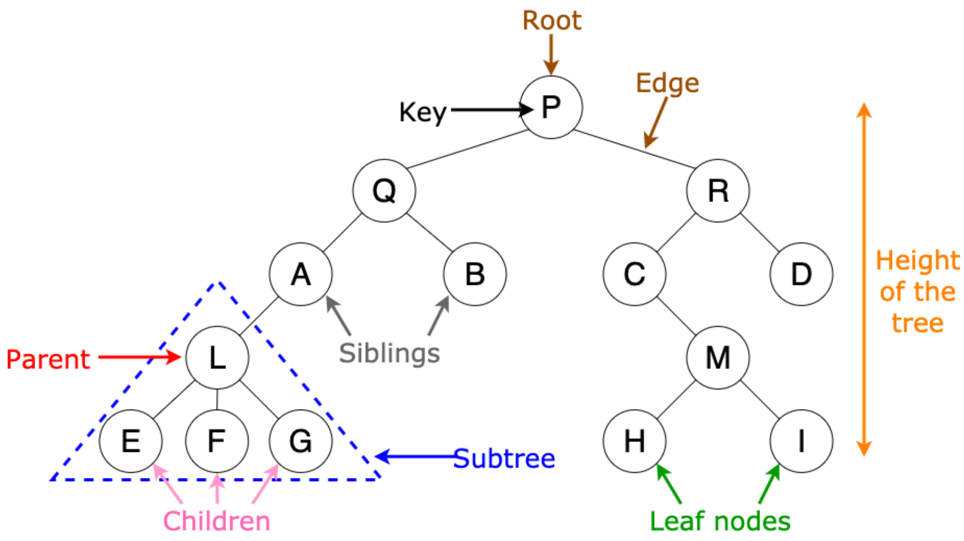
Terminology for a Tree

- **Tree:** A collection of data whose entries have a hierarchical organization
- **Node:** An entry in a tree
- **Root node:** The node at the top
- **Terminal** or **leaf node:** A node at the bottom



Terminology for a Tree (continued)

- **Parent:** The node immediately above a specified node
- **Child:** A node immediately below a specified node
- **Ancestor:** Parent, parent of parent, etc.
- **Descendent:** Child, child of child, etc.
- **Siblings:** Nodes sharing a common parent



Terminology for a Tree (continued)

- **Binary tree:** A tree in which every node has at most two children
- **Depth:** The number of nodes in longest path from root to leaf

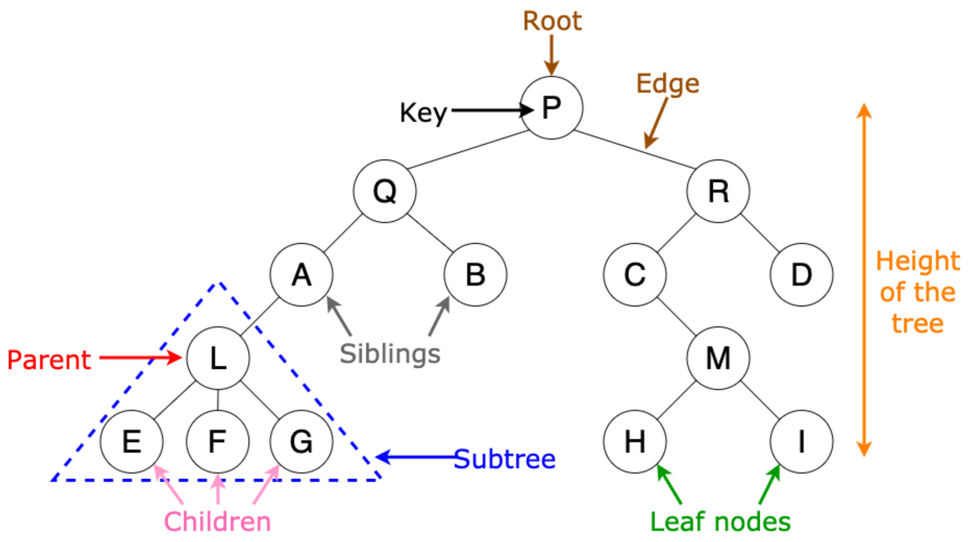
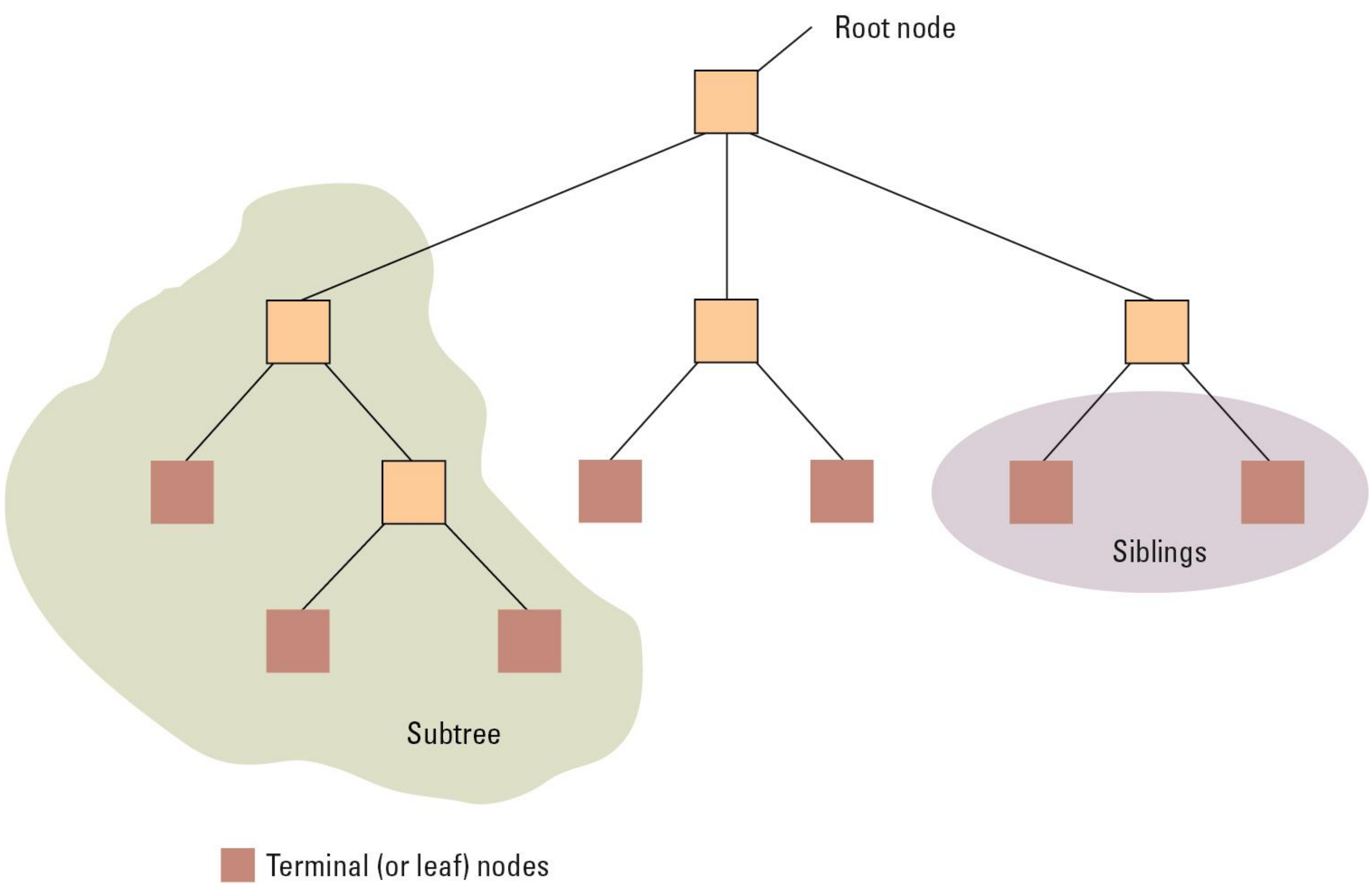
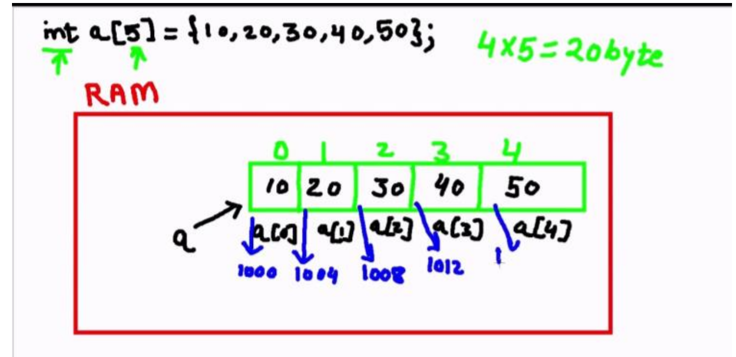


Figure 8.3 Tree terminology



8.2 Related Concepts



- Abstraction
  - Shield *users* (application software) from details of actual data storage

- Static vs. Dynamic Structure
  - Does the shape and size change over time?
- **Pointer**
  - A storage area that contains the address where data is stored

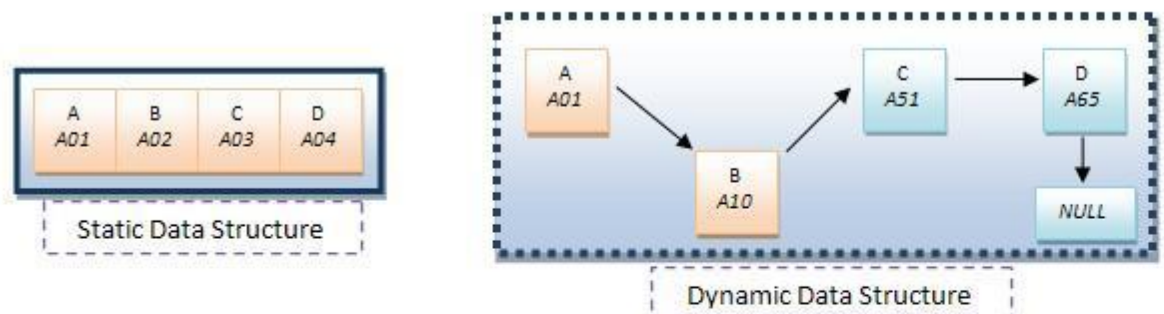
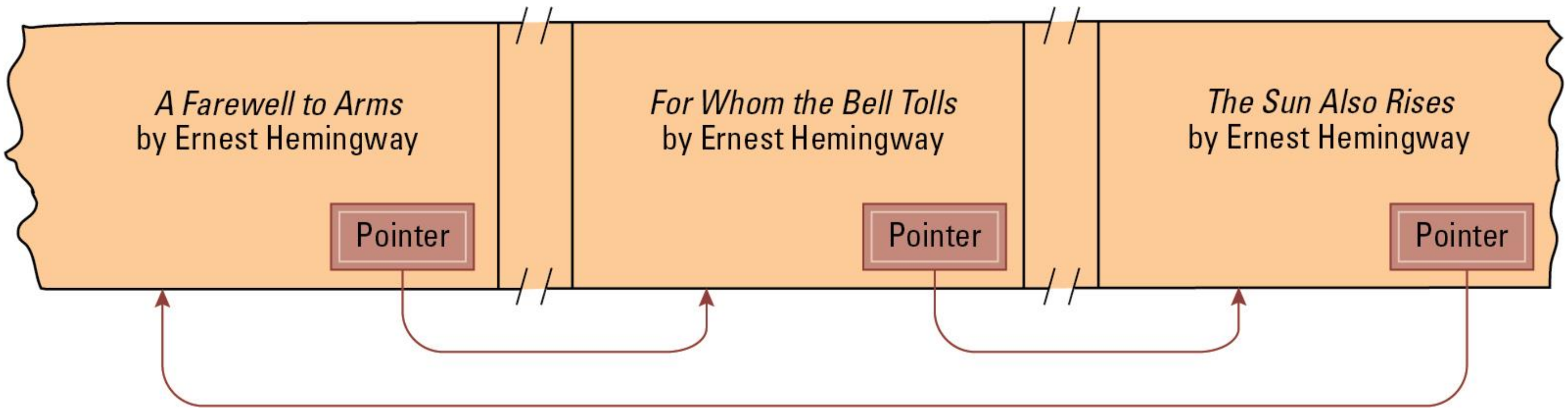


Figure 8.4 Novels arranged by title but linked according to authorship

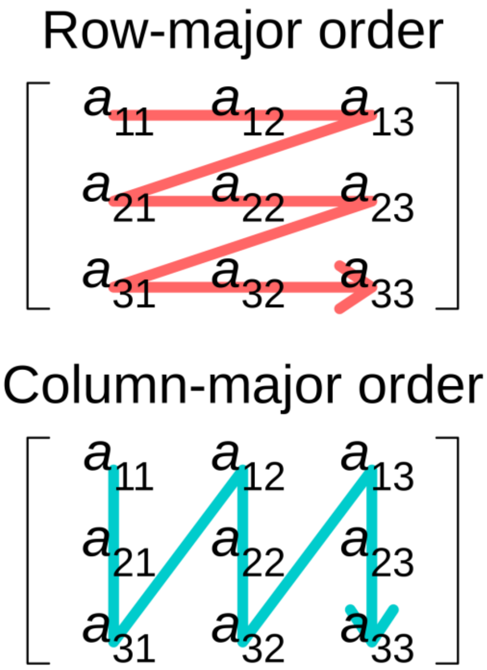


8.3 Implementing Data Structures

- High Level Languages provide certain data structures as primitives
- These data structures are translated into machine-language instructions to manipulate data stored in main memory

Storing Arrays

- Memory address of a particular cell can be computed
- **Row-major order** versus **column major order**
- An Address polynomial describes how to access the data in a certain array element



Storing Arrays

	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

ELEMENTS	ADDRESS	
A[1,1]	200	
A[2,1]	201	Column 1
A[3,1]	202	
A[1,2]	203	
A[2,2]	204	Column 2
A[3,2]	205	
A[1,3]	206	
A[2,3]	207	Column 3
A[3,3]	208	
A[1,4]	209	
A[2,4]	210	Column 4
A[3,4]	211	

Concept:
Column Major Order (CMO)

Pearson

Storing Arrays

	1	2	3	4
1	A[1,1]	A[1,2]	A[1,3]	A[1,4]
2	A[2,1]	A[2,2]	A[2,3]	A[2,4]
3	A[3,1]	A[3,2]	A[3,3]	A[3,4]

Concept:
Row Major Order (RMO)

ELEMENTS	ADDRESS	
A[1,1]	200	Row 1
A[1,2]	201	
A[1,3]	202	
A[1,4]	203	
A[2,1]	204	Row 2
A[2,2]	205	
A[2,3]	206	
A[2,4]	207	
A[3,1]	208	Row 3
A[3,2]	209	
A[3,3]	210	
A[3,4]	211	

Figure 8.5 The array of temperature readings stored in memory starting at address x

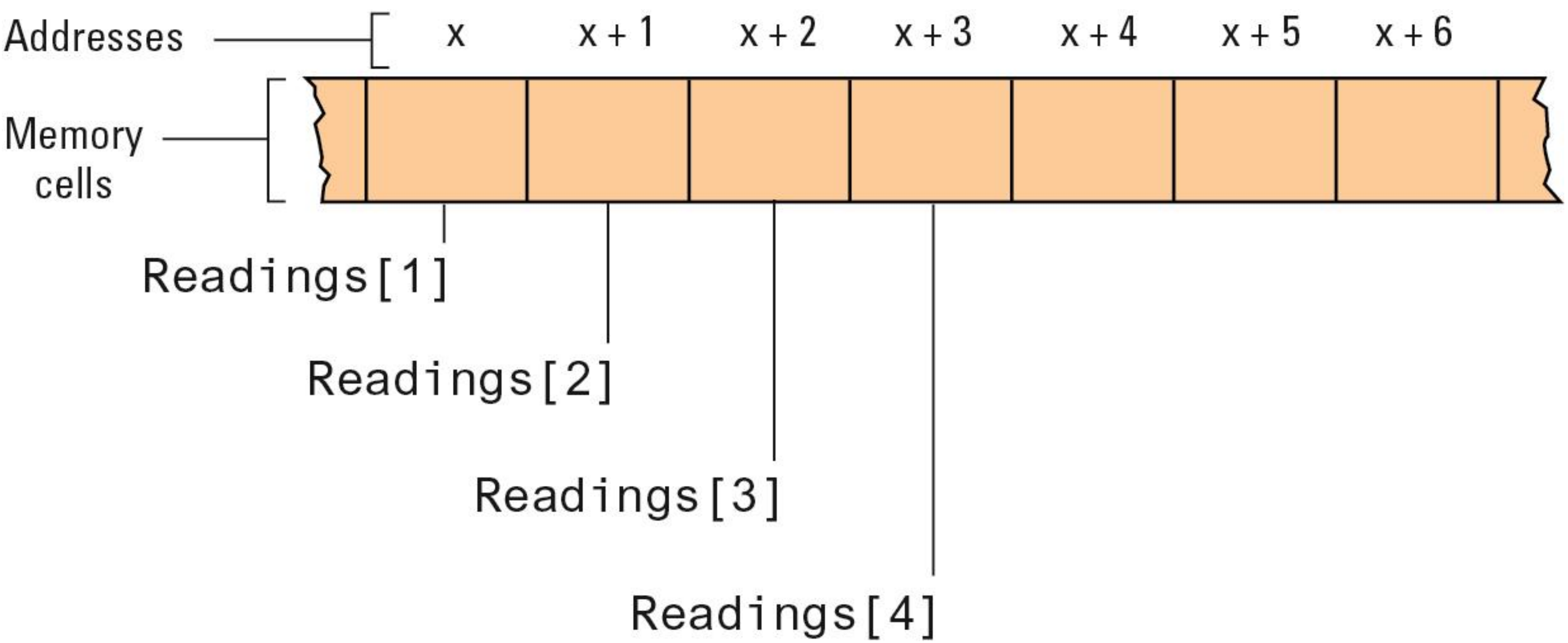
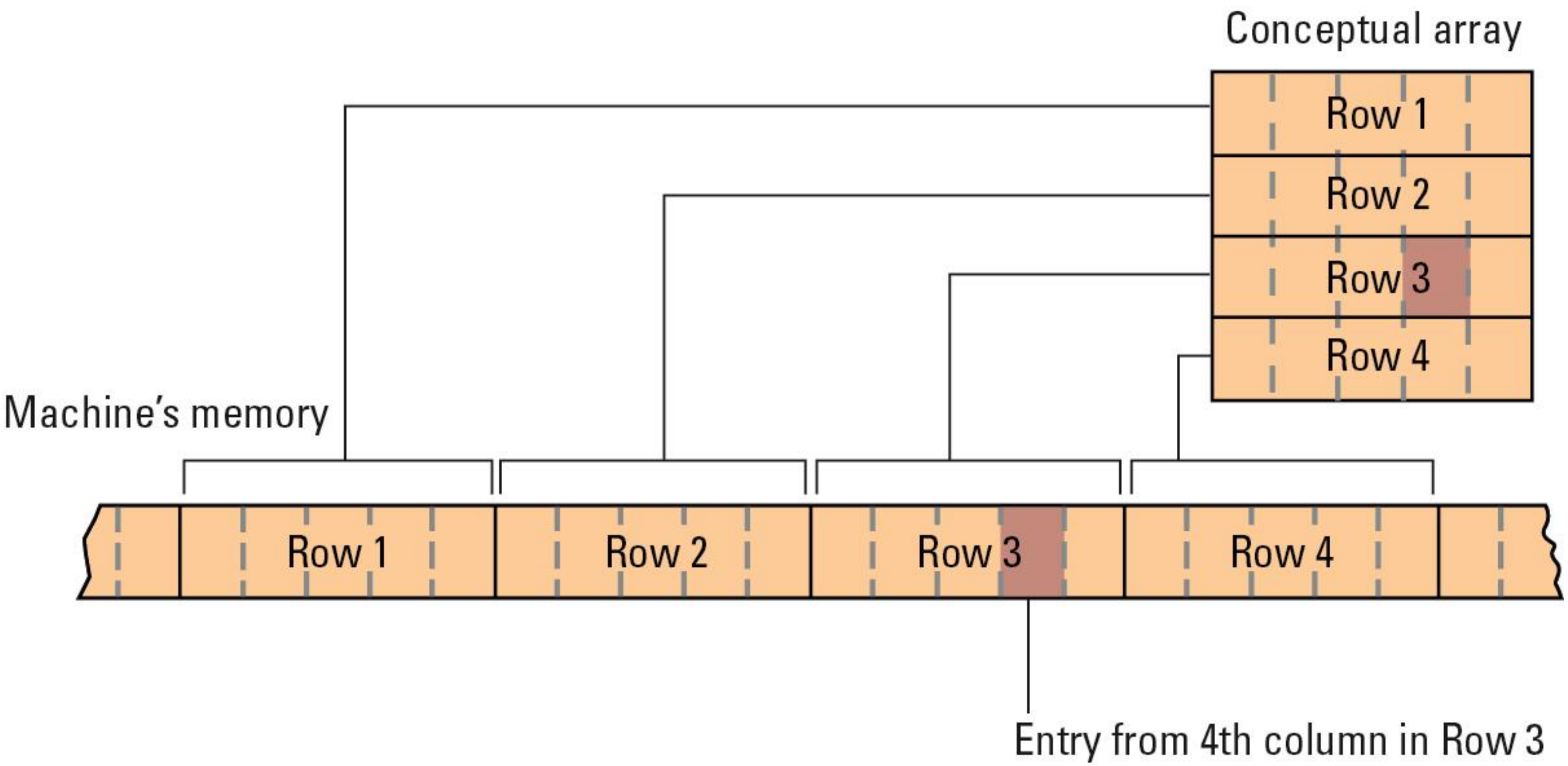


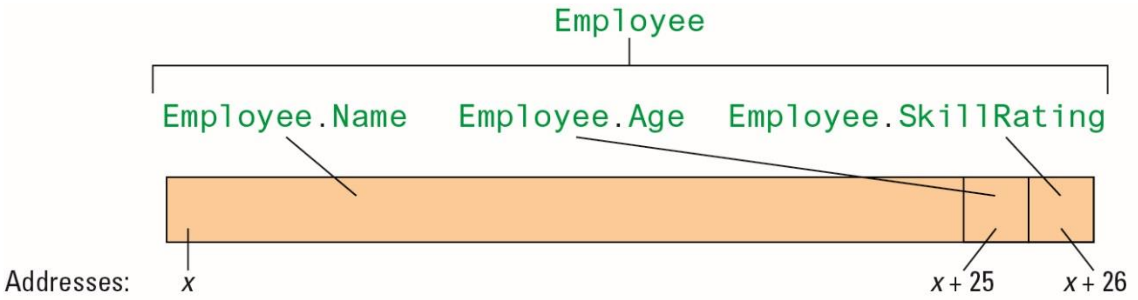
Figure 8.6 A two-dimensional array with four rows and five columns stored in row major order



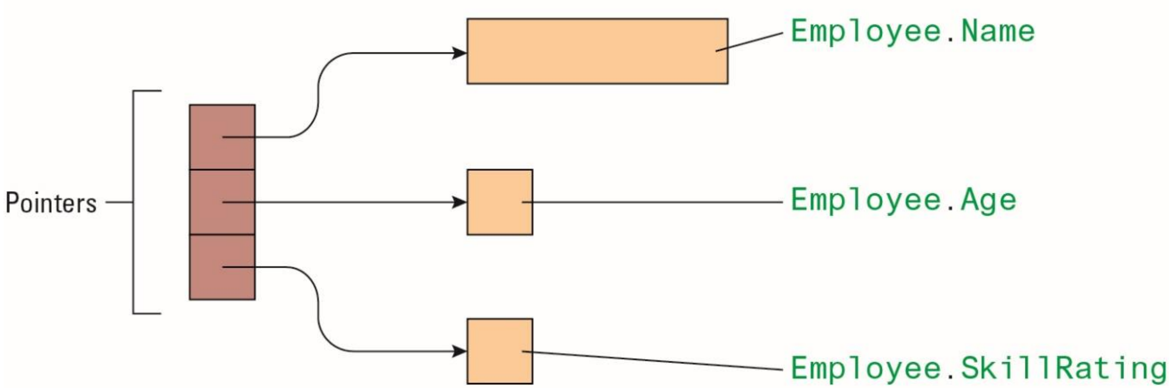
Storing Aggregates

- Fields can be stored one after the other in a contiguous block:
  - Memory cell address of each field can be computed
- Fields can be stored in separate locations identified by pointers

Figure 8.7 Storing the aggregate type Employee



a. Aggregate stored in a contiguous block



b. Aggregate fields stored in separate locations

Storing Lists

- **Contiguous list:** List in which entries are stored in an array
- **Linked list:** List in which entries are linked by pointers
  - **Head pointer:** Pointer to first entry in list
  - **null:** A “non-pointer” value used to indicate end of list

Figure 8.8 Names stored in memory as a contiguous list

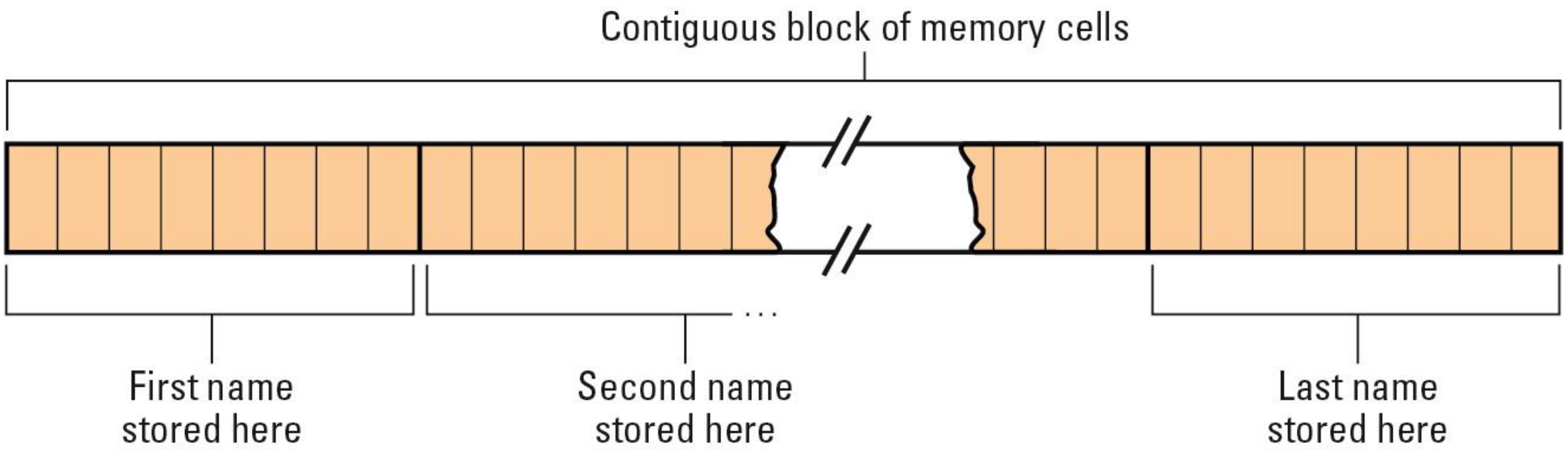


Figure 8.9 The structure of a linked list

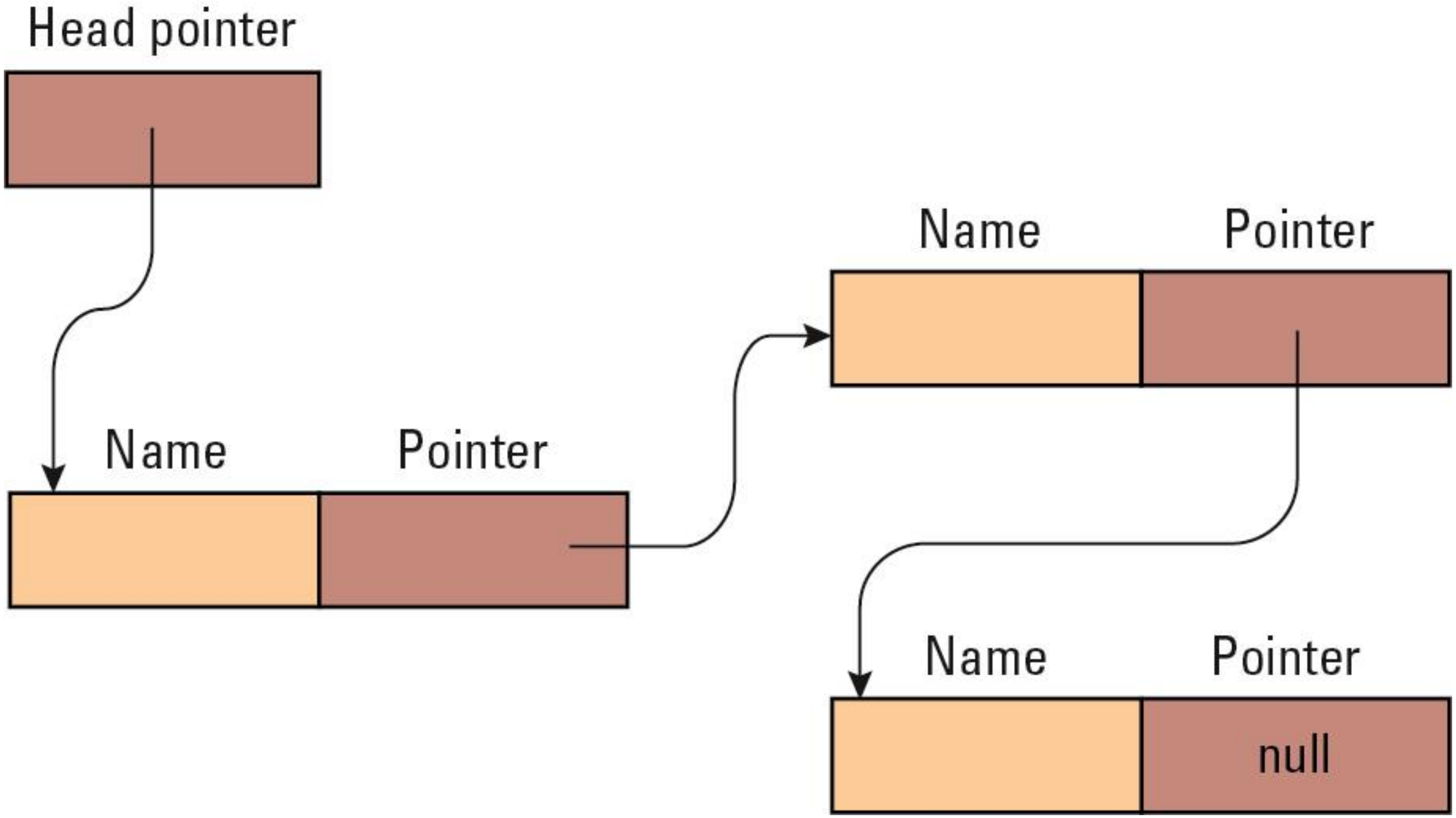


Figure 8.10 Deleting an entry from a linked list

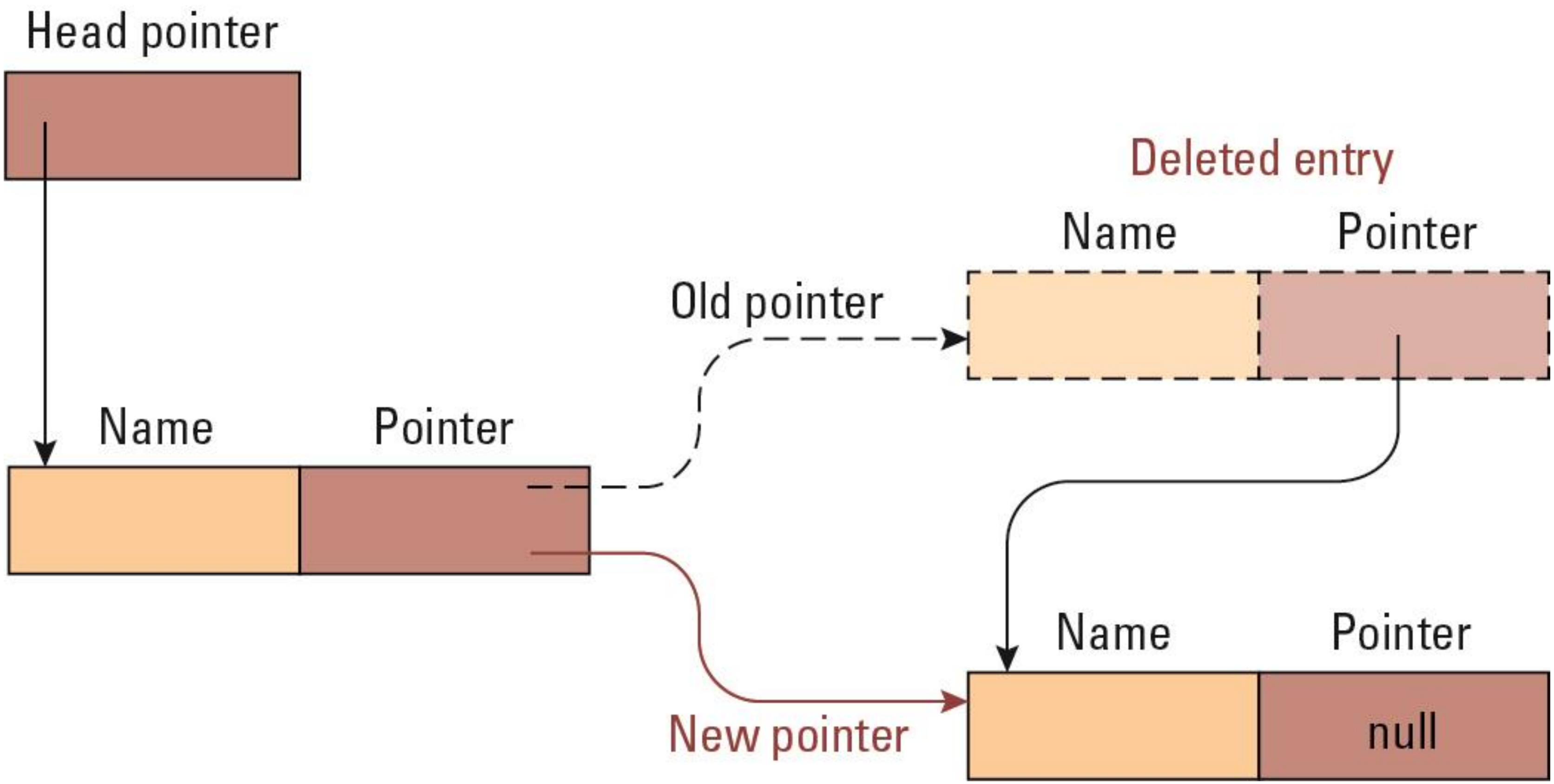
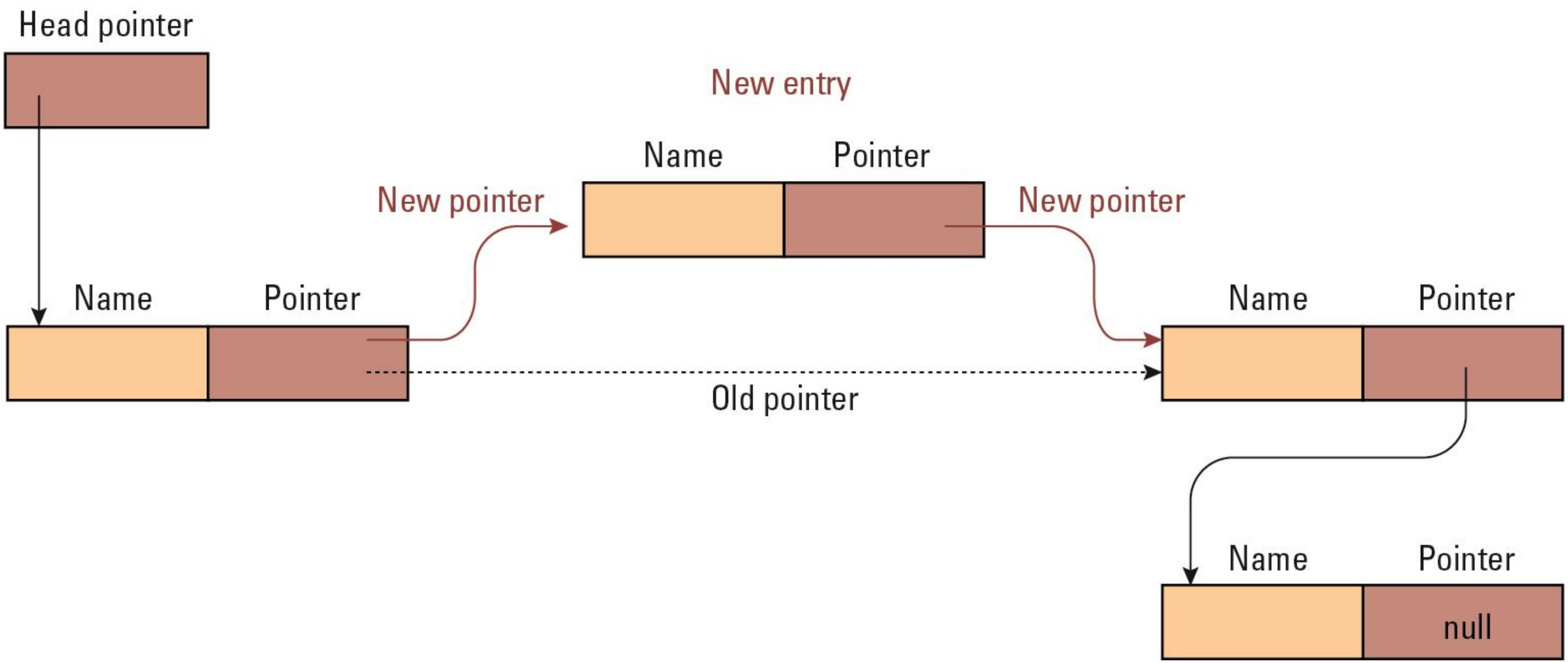


Figure 8.11 Inserting an entry into a linked list



Storing Stacks and Queues

- Stacks usually stored as contiguous lists
- Queues usually stored as **Circular Queues**
  - Stored in a contiguous block in which the first entry is considered to follow the last entry
  - Prevents a queue from crawling out of its allotted storage space

Figure 8.12 A stack in memory

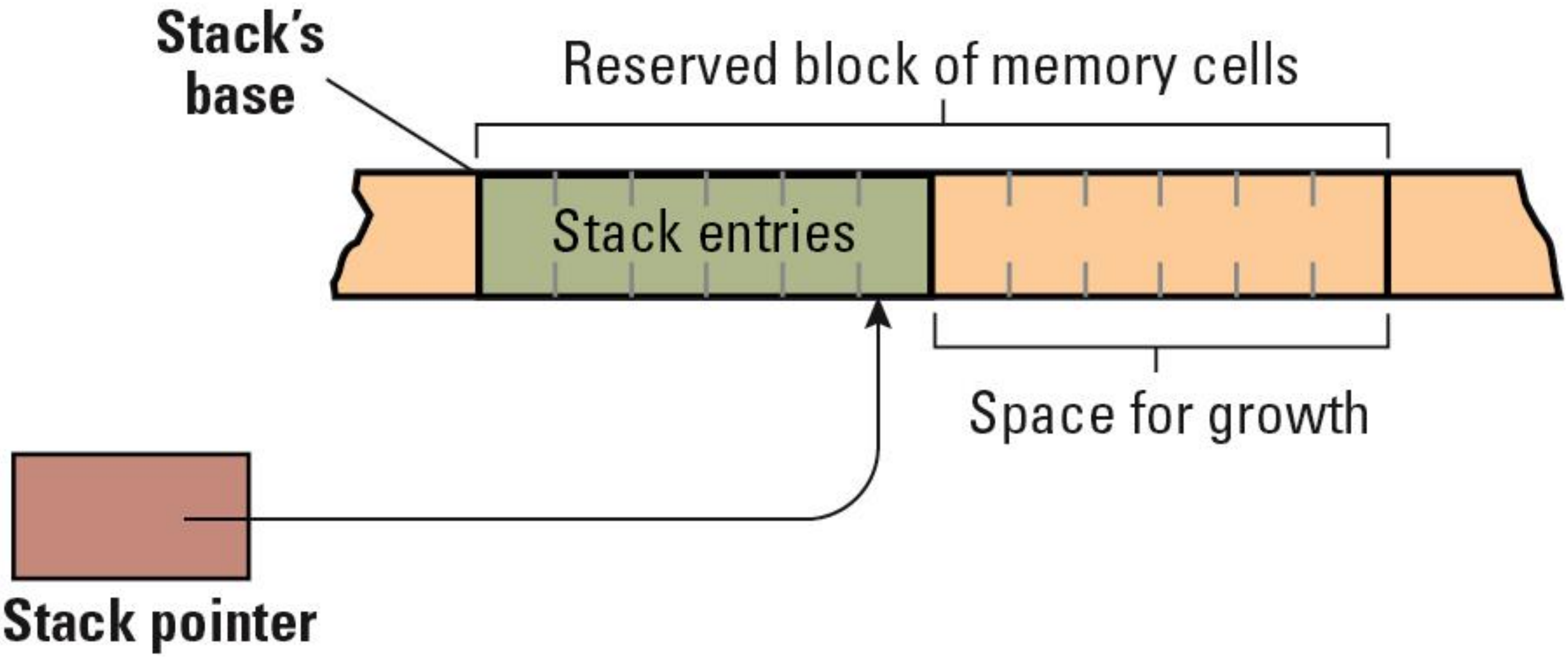


Figure 8.13 A queue implementation with head and tail pointers

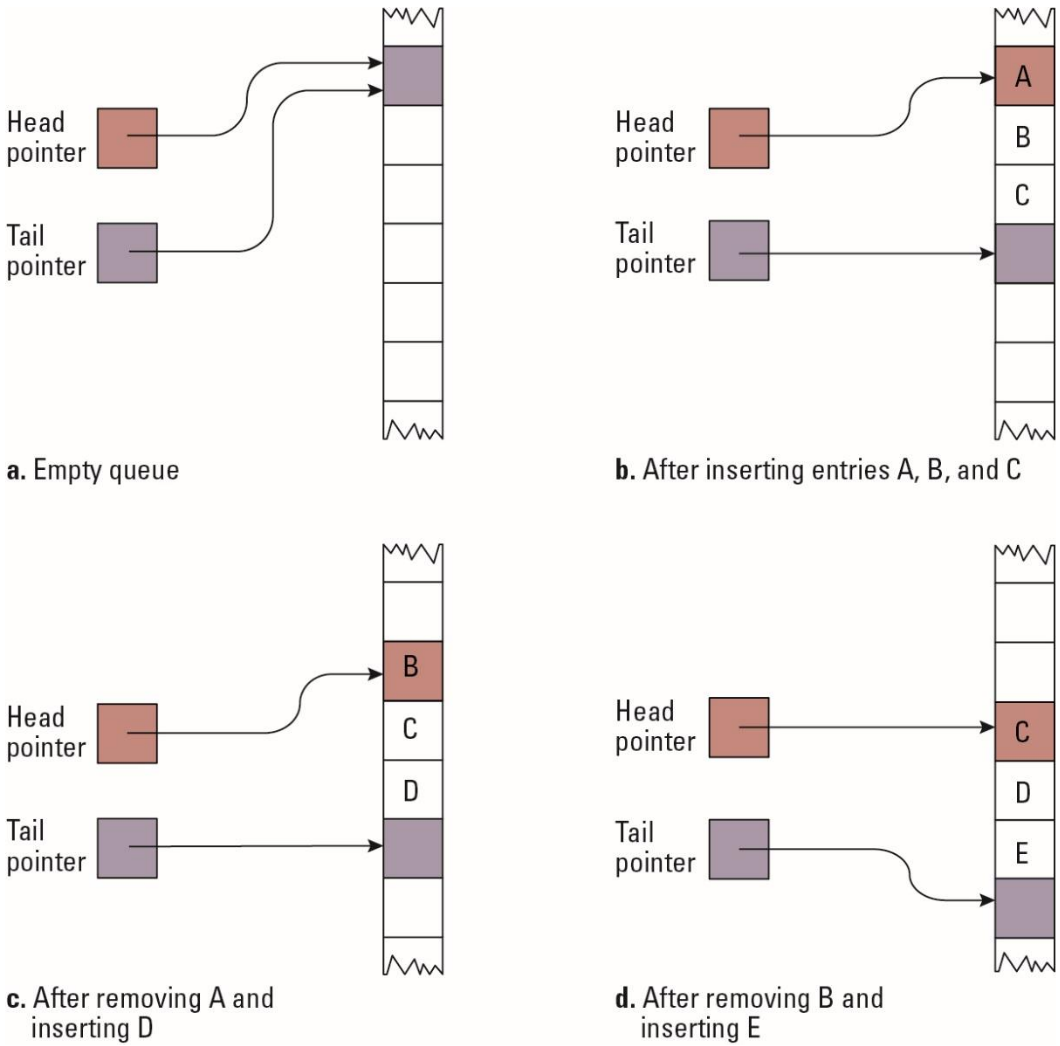
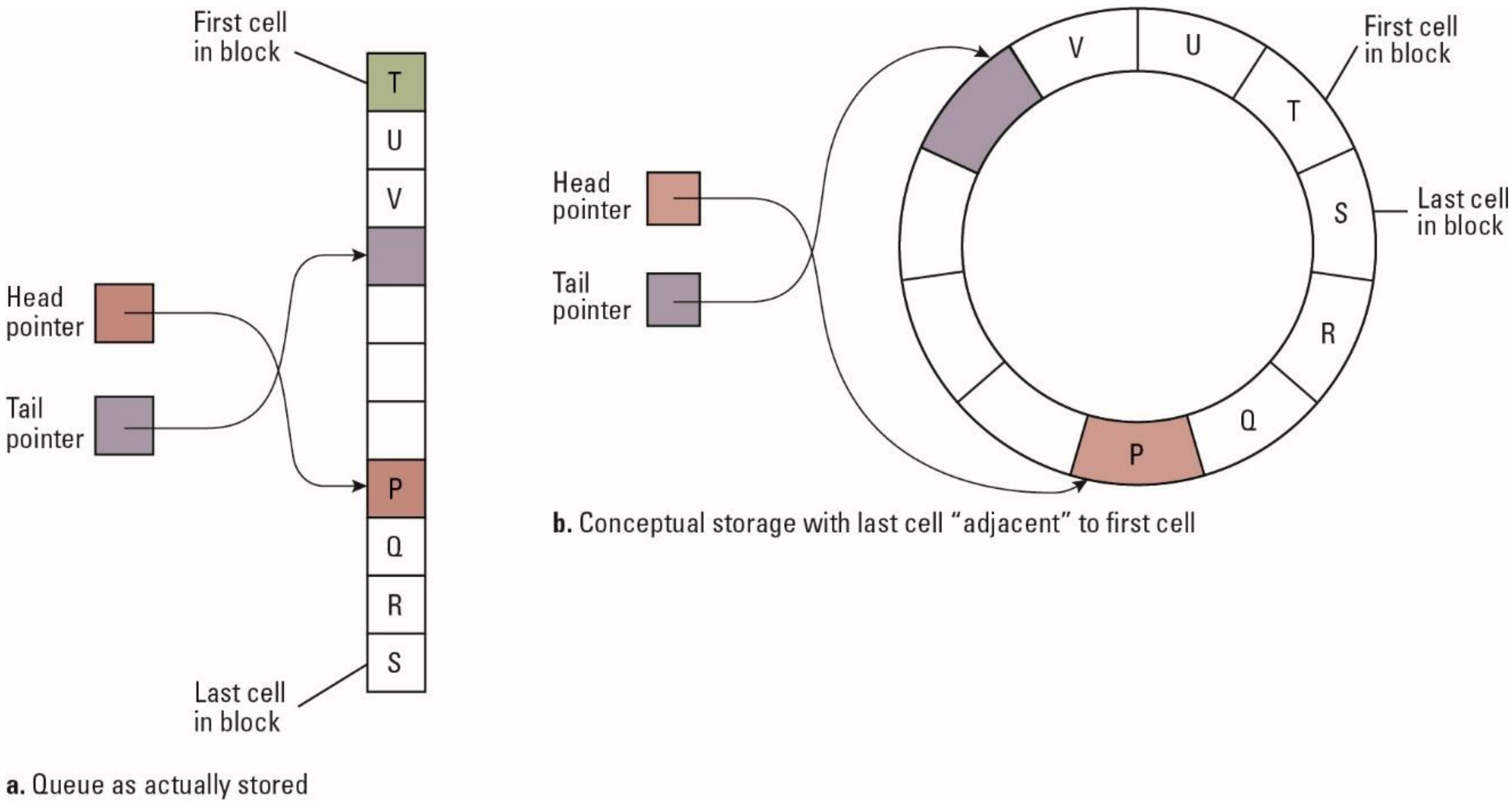


Figure 8.14 A circular queue containing the letters P through V



Storing Binary Trees

- Linked structure
  - Each node = data cells + two child pointers
  - Accessed via a pointer to root node
- Contiguous array structure
  - A[1] = root node
  - A[2],A[3] = children of A[1]
  - A[4],A[5],A[6],A[7] = children of A[2] and A[3]

Figure 8.15 The structure of a node in a binary tree

Figure 8.16 The conceptual and actual organization of a binary tree using a linked storage system

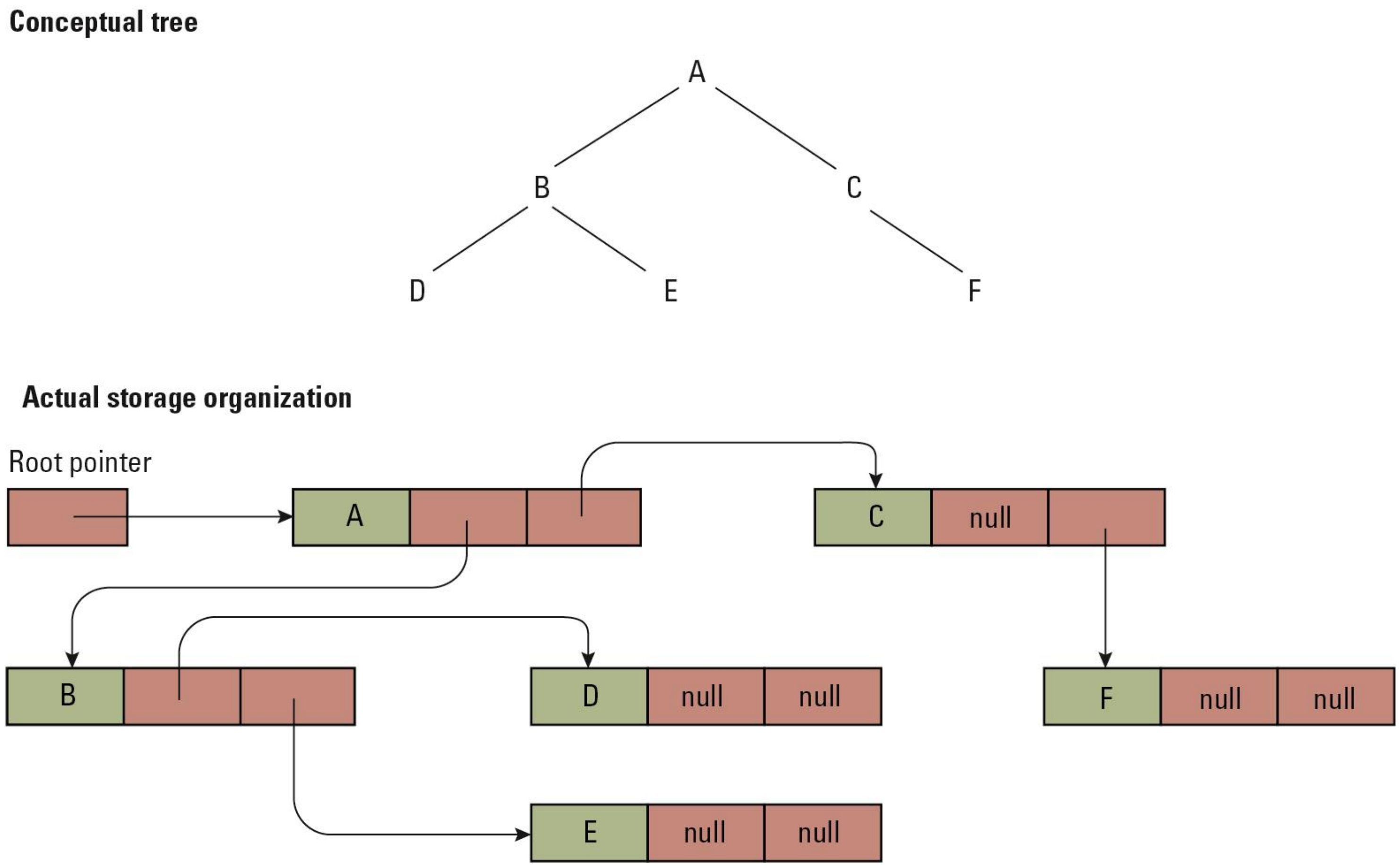


Figure 8.17 A tree stored without pointers

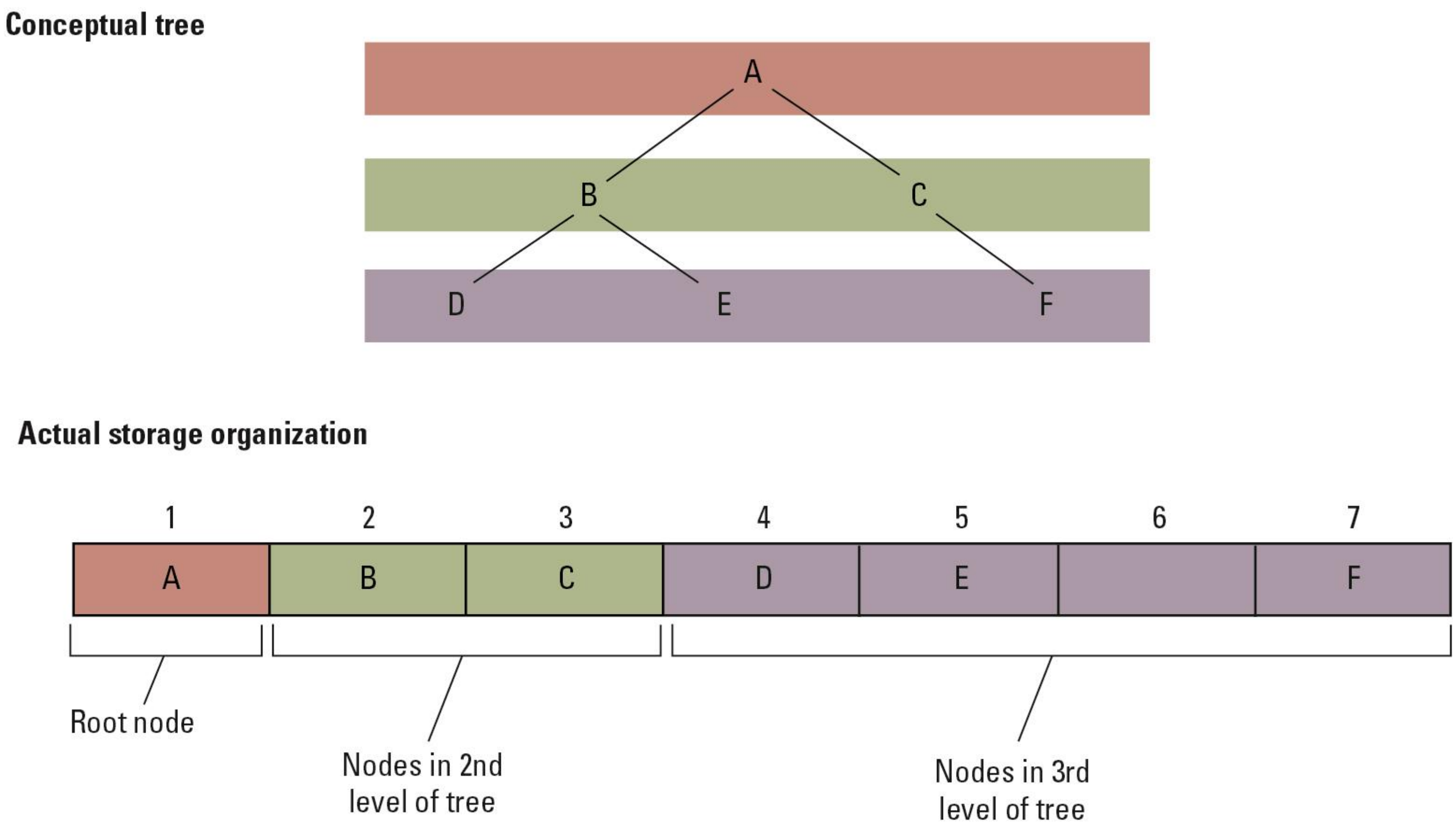
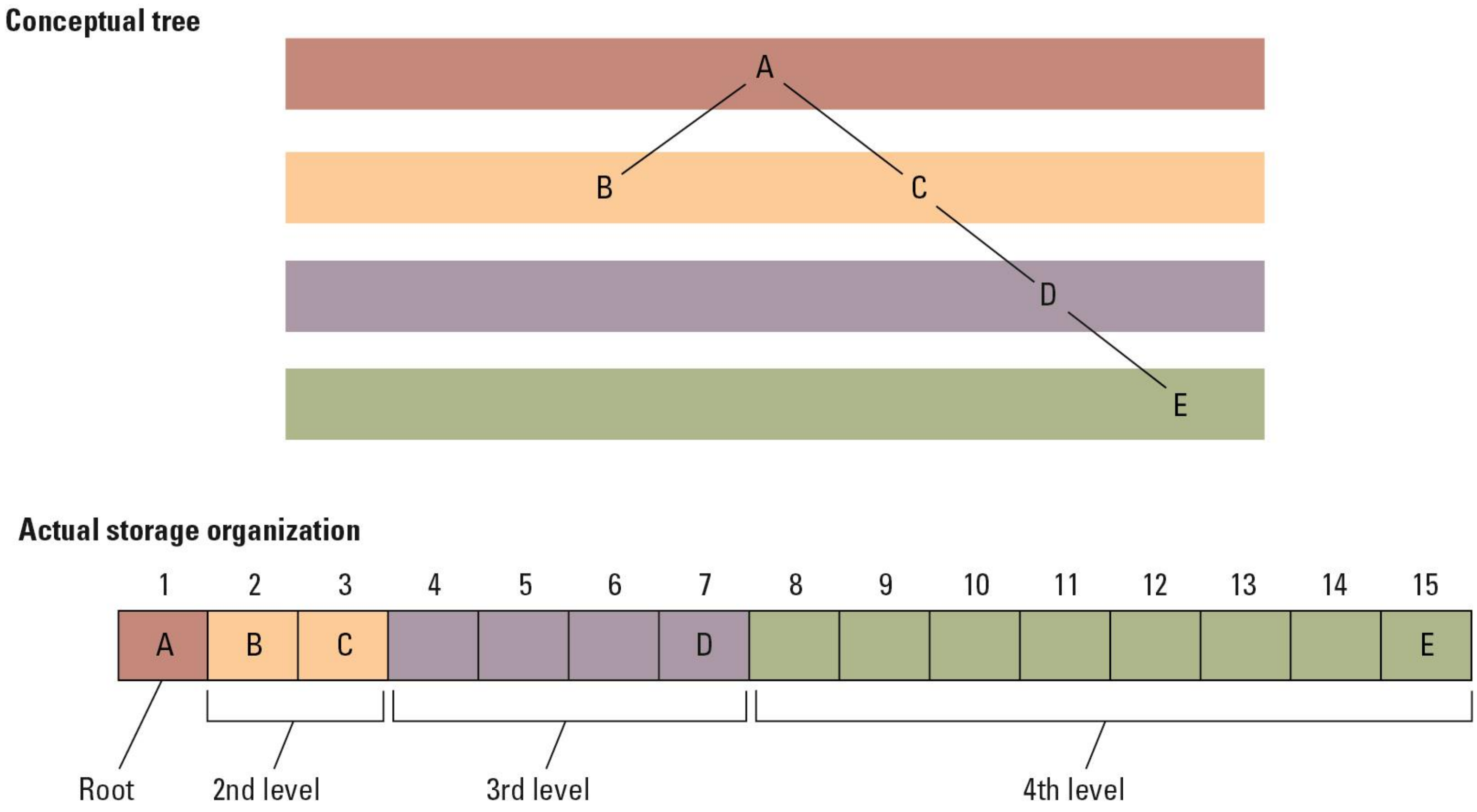


Figure 8.18 A sparse, unbalanced tree shown in its conceptual form and as it would be stored without pointers



Manipulating Data Structures

- Ideally, a data structure should be manipulated solely by pre-defined functions
  - Example: A list typically has a function `insert` for inserting new entries
  - The data structure along with these functions constitutes a complete abstract tool

Pearson

Figure 8.19 A function for printing a linked list

```
def PrintList (List):
    CurrentPointer = List.Head
    while (CurrentPointer is not None):
        print(CurrentPointer.Value)
        CurrentPointer = CurrentPointer.Next
```

8.4 A Short Case Study

Problem:

Construct an abstract tool consisting of a list of names in alphabetical order along with the operations: search, print, and insert.

Figure 8.20 The letters A through M arranged in an ordered tree

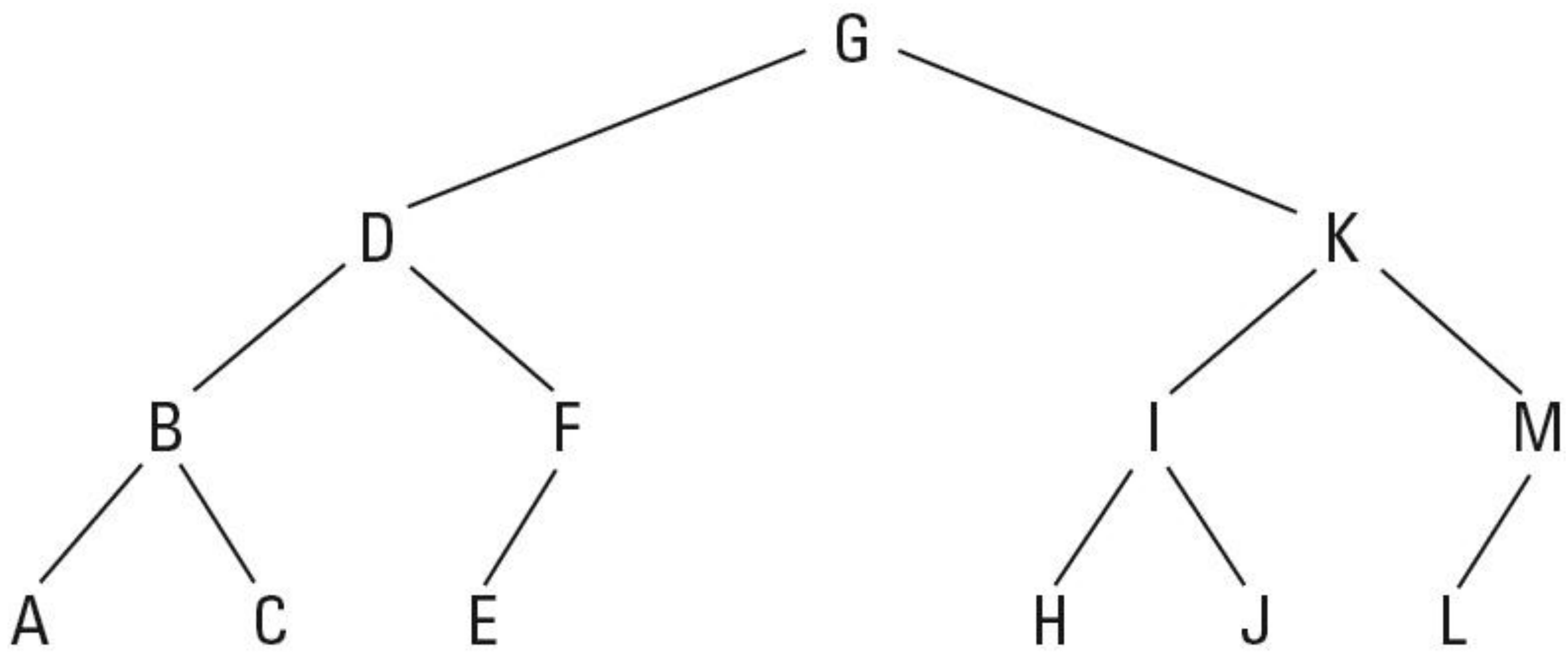


Figure 8.21 The binary search as it would appear if the list were implemented as a linked binary tree

```
def Search (Tree, TargetValue):
    if (Tree is None):
        return None        # Search failed
    elif (TargetValue == Tree.Value):
        return Tree        # Search succeeded
    elif (TargetValue < Tree.Value):
        # Continue search in left subtree
        return Search(Tree.Left, TargetValue)
    elif (TargetValue > Tree.Value):
        # Continue search in right subtree
        return Search(Tree.Right, TargetValue)
```

Figure 8.22 The successively smaller trees considered by the function in Figure 8.21 when searching for the letter J

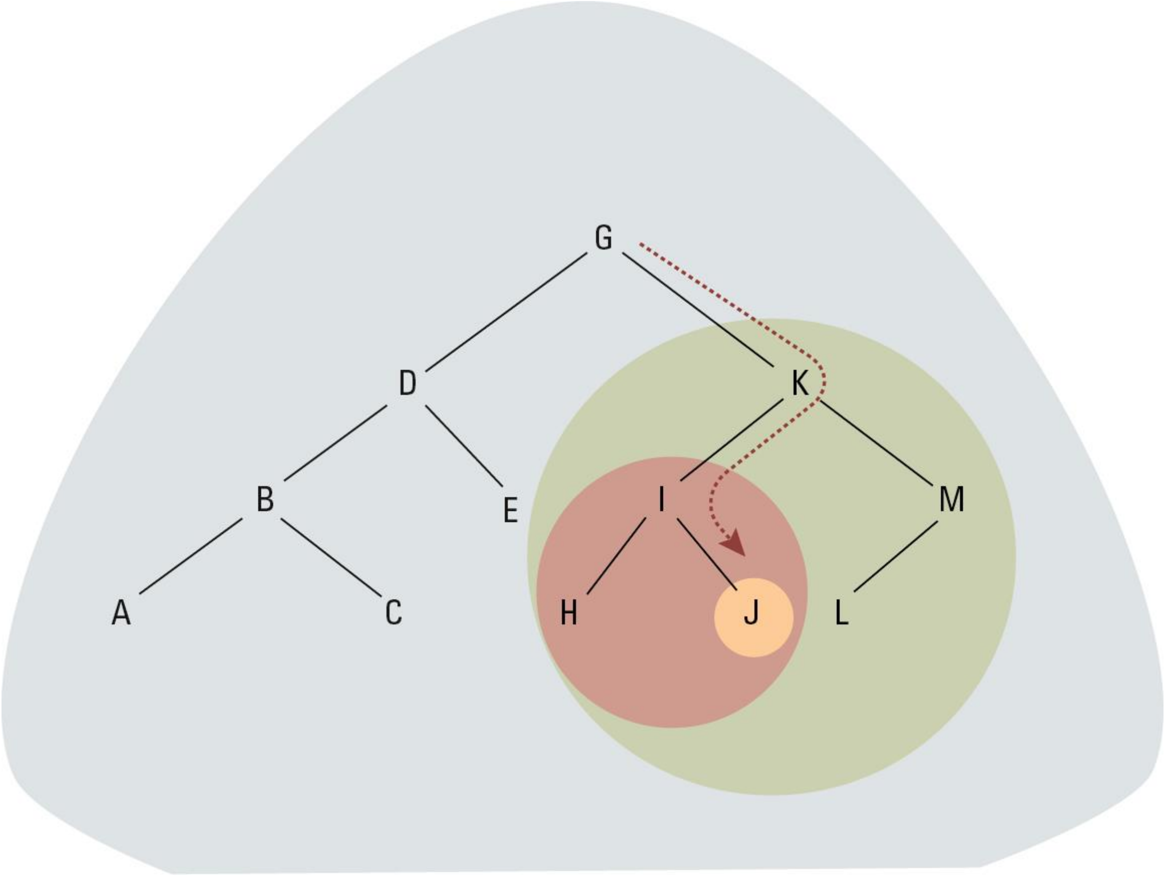


Figure 8.23 Printing a search tree in alphabetical order

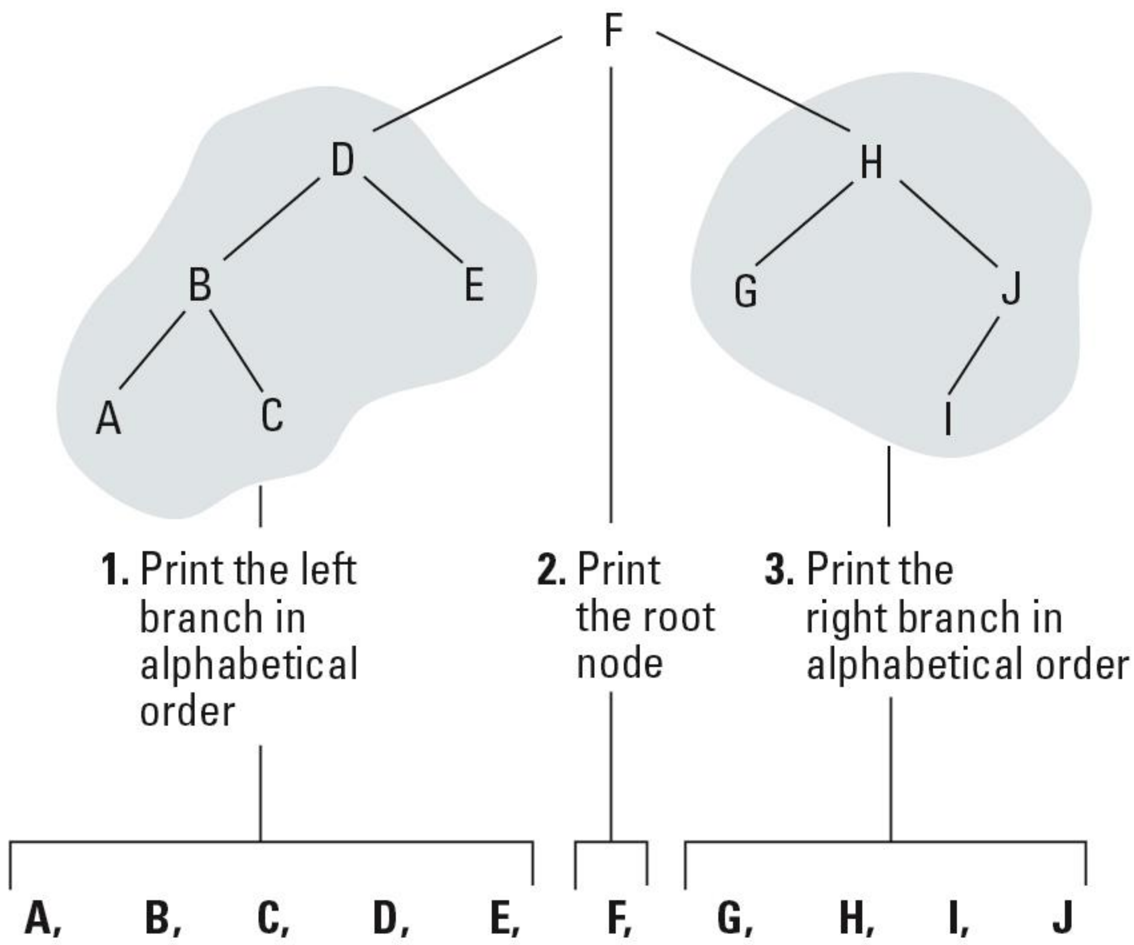


Figure 8.24 A function for printing the data in a binary tree

```
def PrintTree (Tree):
    if (Tree is not None):
        PrintTree(Tree.Left)
        print(Tree.Value)
        PrintTree(Tree.Right)
```

Pearson

Figure 8.25 Inserting the entry M into the list B, E, G, H, J, K, N, P stored as a tree

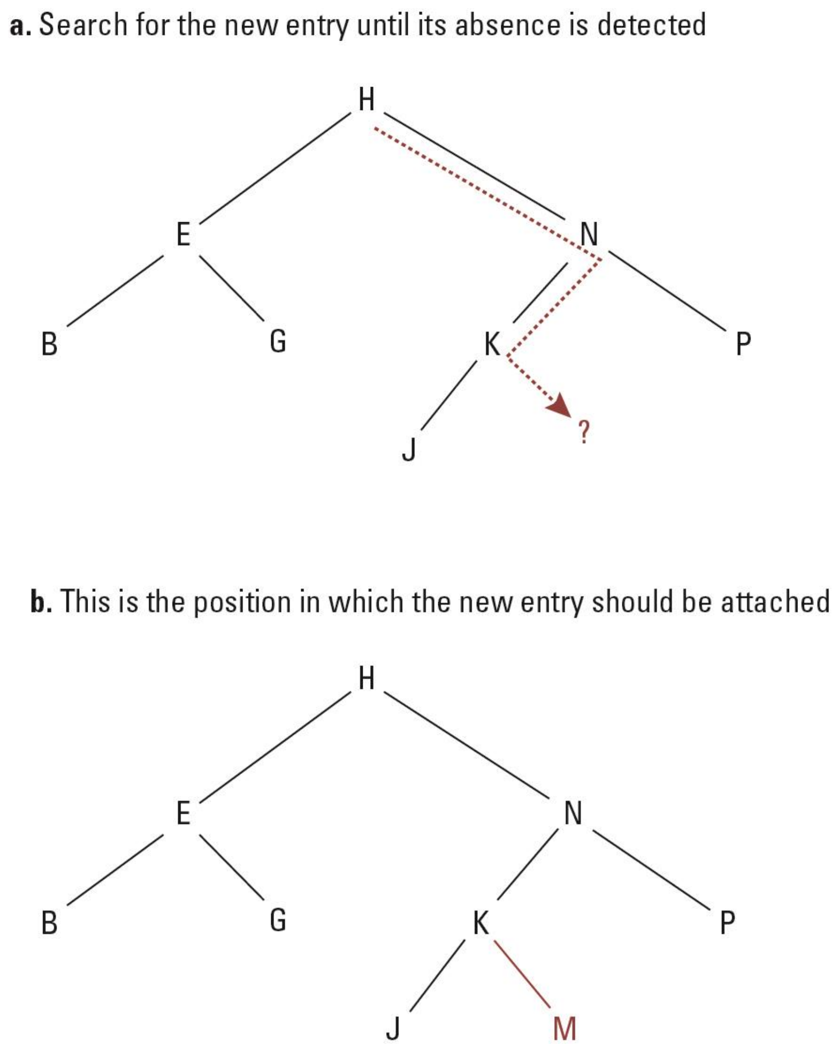


Figure 8.26 A function for inserting a new entry in a list stored as a binary tree

```
def Insert(Tree, NewValue):
    if (Tree is None):
        # Create a new leaf with NewValue
        Tree = TreeNode()
        Tree.Value = NewValue
    elif (NewValue < Tree.Value):
        # Insert NewValue into the left subtree
        Tree.Left = Insert(Tree.Left, NewValue)
    elif (NewValue > Tree.Value):
        # Insert NewValue into the right subtree
        Tree.Right = Insert(Tree.Right, NewValue)
    else:
        # Make no change
    return Tree
```

8.5 Customized Data Types

Primitive Types:

integer, float, character, Boolean

A programmer can define customized data types to meet the needs of a particular application

User-defined Data Type

- Use an aggregate structure to define new type, in C:

```
struct EmployeeType
{
  char     Name[25];
  int      Age;
  real     SkillRating;
}
```

- Use the new type to define variables:

```
struct EmployeeType DistManager, SalesRep1;
```

Abstract Data Type

- A user-defined data type that can include both data (representation) and functions (behavior)
- Example:

```
interface StackType
{
    public int pop();
    public int push(int item);
    public boolean isEmpty();
    public boolean isFull();
}
```

8.6 Classes and Objecs

- Class: an abstract data type with extra features
  - Properties can be inherited
  - Constructor methods to initialize new objects
  - Contents can be encapsulated
- Object: an instance of a class

Figure 8.27 A stack of integers implemented in Java and C#

```
class StackOfIntegers implements StackType
{
    private int[] StackEntries = new int[20];
    private int StackPointer = 0;

    public void push(int NewEntry)

    {   if (StackPointer < 20)
            StackEntries[StackPointer++] = NewEntry;
    }

    public int pop()
    {   if (StackPointer > 0) return StackEntries[--StackPointer];
        else return 0;
    }

    public boolean isEmpty()
    {   return (StackPointer == 0);
    }

    public boolean isFull()
    {   return (StackPointer >= MAX);
    }
}
```

8.7 Pointers in Machine Language

- **Immediate addressing**: Instruction contains the data to be accessed
- **Direct addressing**: Instruction contains the address of the data to be accessed
- **Indirect addressing**: Instruction contains the location of the address of the data to be accessed

Figure 8.28 Our first attempt at expanding the machine language in Appendix C to take advantage of pointers

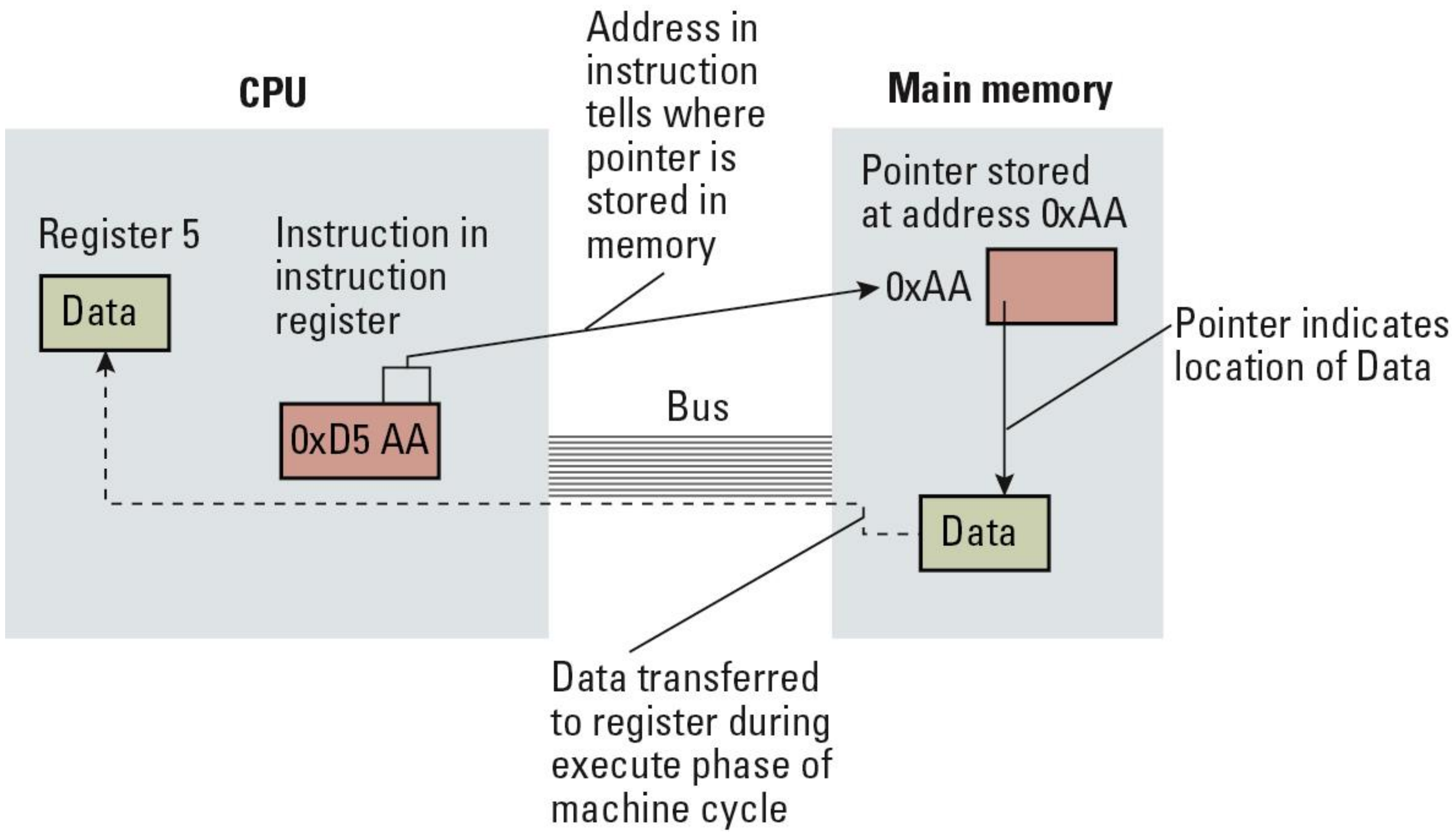


Figure 8.29 Loading a register from a memory cell that is located by means of a pointer stored in a register